Compact Graph Representations and Parallel Connectivity Algorithms for Massive Dynamic Network Analysis

Kamesh Madduri
Computational Research Division
Lawrence Berkeley National Laboratory
Berkeley, USA 94703

David A. Bader
College of Computing
Georgia Institute of Technology
Atlanta, USA 30332

Abstract

Graph-theoretic abstractions are extensively used to analyze massive data sets. Temporal data streams from socio-economic interactions, social networking web sites, communication traffic, and scientific computing can be intuitively modeled as graphs. We present the first study of novel high-performance combinatorial techniques for analyzing large-scale information networks, encapsulating dynamic interaction data in the order of billions of entities. We present new data structures to represent dynamic interaction networks, and discuss algorithms for processing parallel insertions and deletions of edges in small-world networks. With these new approaches, we achieve an average performance rate of 25 million structural updates per second and a parallel speedup of nearly 28 on a 64-way Sun UltraSPARC T2 multicore processor, for insertions and deletions to a small-world network of 33.5 million vertices and 268 million edges. We also design parallel implementations of fundamental dynamic graph kernels related to connectivity and centrality queries. Our implementations are freely distributed as part of the open-source SNAP (Small-world Network Analysis and Partitioning) complex network analysis framework.

1. Introduction

Graphs are a fundamental abstraction for modeling and analyzing data, and are pervasive in real-world applications. Transportation networks (road and airline traffic), socio-economic interactions (friendship circles, organizational hierarchies, online collaboration networks), and biological systems (food webs, protein interaction networks) are a few examples of data that can be naturally represented as graphs. Understanding the dynamics and evolution of real-world networks is a "Grand Challenge" science and mathematics problem [26]. Networks are being studied extensively from an empirical perspective, and this has led to a variety of models to characterize their topology and evolution. Quite surprisingly, it has been shown that several real-world systems such as the Internet, social interactions, and biological networks exhibit common structural features such as a low graph diameter, unbalanced degree distributions, self-similarity, and the presence of dense sub-graphs [1, 21]. This is generally referred to as the *small-world* phenomenon, and these topological characteristics can be exploited for fast and efficient algorithm design.

We tackle two challenging aspects of small-world graph analysis in this paper: the *problem scale*, and the *temporal nature* of interaction networks. Due to technological advances related to the Internet, sensor networks, experimental devices, biological sequencing, and science and engineering simulations, we are experiencing an explosion of informatics data on several fronts. The network representations of these data sets may have millions, or even billions, of entities. We need new algorithms that scale with the problem size, and are optimized for efficient execution on emerging computer architectures such as multicore systems. In prior work, we present efficient implementations for fundamental graph problems and kernels such as list ranking [3], breadth-first search and *st*-connectivity [4], shortest paths [19], minimum spanning trees [2], and connected components [3]. We exploit the small-world topology of the network to design new parallelization strategies, and demonstrate the ability to process massive graphs in the order of billions of entities [6]. However, the previous research has primarily focused on a static representation of the network. We are increasingly observing dynamically evolving networks in real-world systems, with heterogeneity in the data content, and variable data quality. The challenge here is to design efficient implementations and algorithms that can handle structural updates to these complex networks, while solving graph queries with as little additional work as possible, and minimizing the size of auxiliary data structures.

This is the first paper to identify, design, and implement fundamental graph data structures and kernels for analyzing massive dynamic networks on parallel systems. Our main results are summarized here:

- We design a new hybrid data structure for dynamic small-world graphs that efficiently processes updates to *high-degree* vertices. We also experiment with several data representations for a stream of structural updates (insertions and deletions of edges), and demonstrate

scalable parallel performance on massive networks. For instance, we achieve an average performance of 25 million structural updates per second, and a parallel speedup of nearly 28 on the 64-way Sun UltraSPARC T2 multicore processor (8 cores, and 8 threads per core), for construction, insertions, and deletions to a small-world network of 33.5 million vertices and 268 million edges.

- To address connectivity problems in dynamic networks, we show that a simple implementation of the link-cut tree helps us process queries in time proportional to the diameter of the network. We design parallel approaches for tree construction, updates, as well as query processing. The implementations scale quite well on multicore architectures. For instance, constructing a link-cut tree for a network of 10 million vertices and 84 million edges takes about 3 seconds on the UltraSPARC T2, and we can process connectivity queries on this network at the rate of 7.3 million per second.
- For answering path-related and centrality queries, we design an efficient induced subgraph kernel, graph traversal algorithm, and formulate a betweenness centrality algorithm to deal with time-stamps on the edges. We demonstrate scalable parallel performance for small-world instances, achieving a parallel speedup of 13.1 for breadth-first search on a massive dynamic network of 500 million vertices and 4 billion edges, on the IBM p5 570 symmetric multiprocessor system.

The techniques discussed in this paper directly impact several important real-world applications, such as graph database operations for biological networks (approximate matching [25], summarization, and clustering), queries on massive dynamic interaction data sets in intelligence and surveillance [9, 18], and Web algorithms. The parallel implementations of these dynamic graph kernels are freely available as part of the open-source SNAP complex network analysis framework [6].

This paper is organized as follows. We give a brief introduction to complex network analysis, and discuss prior work related to dynamic graph algorithms in Section 1.1. To evaluate our new algorithms, we conduct an extensive experimental study, and the test setup is detailed in Section 1.2. Sections 2 and 3 present details of our new dynamic graph representations and kernels respectively, and we also analyze their performance on several emerging multithreaded and multicore architectures. We conclude with a discussion of research challenges in this area in Section 4.

1.1. Related Work

For tractable analysis of massive temporal data sets, we need new algorithms and software implementations that supplement existing approaches for processing static graphs. Innovative algorithmic techniques from the areas of streaming algorithms, dynamic graph algorithms, social network analysis, and parallel algorithms for combinatorial problems are closely related to the problems we are trying to solve, and we give a brief overview of research in these domains.

Data Stream and Classical Graph Algorithms. The data stream model [20] is a powerful abstraction for the statistical mining of massive data sets. The key assumptions in this model are that the input data stream may be potentially unbounded and transient, the computing resources are sub-linear in data size, and queries may be answered with only one (or a few) pass(es) over the input data. The bounded memory and computing-resource assumption makes it infeasible to answer most streaming queries exactly, and so approximate answers are acceptable. Effective techniques for approximate query processing include sampling, batch-processing, sketching, and synopsis data structures. Complementing data stream algorithms, a graph or network representation is a convenient abstraction in many applications – unique data entities are represented as vertices, and the interactions between them are depicted as edges. The vertices and edges can further be typed, classified, or assigned attributes based on relational information from the heterogeneous sources. Analyzing topological characteristics of the network, such as the vertex degree distribution, centrality and community structure, provides valuable insight into the structure and function of the interacting data entities. Common analysis queries on the data set are naturally expressed as variants of problems related to graph connectivity, flow, or partitioning.

Since classical graph problems are typically formulated in an imperative, state-based manner, it is hard to adapt existing algorithms to the data stream model. New approaches have been proposed for solving graph algorithms in the streaming model [15], but there are no known algorithms to solve fundamental graph problems in sub-linear space and a constant number of data stream passes. The semi-streaming model is a relaxation to the classical streaming model, that allows $O(n \text{ polylog } n)$ space and multiple passes over data. This is a simpler model for solving graph problems and several recent successes have been reported [11]. However, this work is far from complete; we require faster exact and approximate algorithms to analyze peta- and exascale data sets and to experimentally evaluate the proposed semi-streaming algorithms on current architectures.

Dynamic Graph Algorithms. While the focus of streaming algorithms is on processing massive amounts of data assuming limited computational and memory resources, the research area of dynamic graph algorithms [12] in graph theory deals with work-efficient algorithms for temporal graph problems. The objective of dynamic graph algorithms is to efficiently maintain a desired graph property (for

instance, connectivity, or the spanning tree) under a dynamic setting, i.e. allowing periodic insertion and deletion of edges, and edge weight updates. A dynamic graph algorithm should process queries related to a graph property faster than recomputing from scratch, and also perform topological updates quickly. The dynamic tree problem is a key kernel [27] in several dynamic graph algorithms; Eppstein et al.'s sparsification [13] and Henzinger et al.'s randomization methods [14] are novel algorithmic techniques proposed for processing temporal graphs. A *fully dynamic* algorithm handles both edge insertions and deletions, whereas a *partially dynamic* algorithm handles only one of them. Dynamic graph algorithms have been designed for the all-pairs shortest paths, maximum flow, minimum spanning forests and other graph applications [10]. Recent experimental studies [28] have evaluated the performance trade-offs involved in some of these kernels and techniques.

Large-scale Network Analysis and Parallel Computing. The analysis of complex interaction data is an active research area in the social science and statistical physics communities. Modeling of networks (such as the *small-world* network model) and quantitative measures to better understand these complex networks (for instance, identification of influential entities, communities, and anomalous patterns) are well-studied [21]. Computations involving these sparse real-world graphs only manage to achieve a tiny fraction of the peak system performance on the majority of current computing platforms. The primary reason is that sparse graph analysis tends to be highly memory-intensive: codes typically have a large memory footprint, exhibit low degrees of spatial and temporal locality in their memory access patterns (compared to other workloads), and there is very little computation to hide the latency to memory accesses. Further, on current workstations, it is infeasible to do exact in-core computations on large-scale graphs (by *large-scale*, we refer to graphs where the number of vertices and edges are in the range of hundreds of millions to tens of billions) due to the limited physical memory and running time constraints. In such cases, emerging manycore and multithreaded architectures with a significant amount of shared physical memory (say, 4-32 GB or more) are a natural platform for the design of efficient multithreaded graph algorithms and processing massive networks. For instance, recent experimental studies on breadth-first search for large-scale sparse random graphs show that a parallel in-core implementation is two orders of magnitude faster than an optimized external memory implementation [4]. Parallel multicore and manycore processors are the building blocks for petascale supercomputers, and are now ubiquitous in home computing also.

1.2. Experimental Setup

In our experimental studies, we use the Recursive MATrix (R-MAT) [8] random graph generation algorithm to generate input data sampled from a Kronecker product that are representative of real-world networks with a small-world topology. The R-MAT generator creates directed as well as undirected graphs with $n = 2^k$ vertices and m edges, and allows shaping parameters. We set these parameters a, b, c, and d to 0.6, 0.15, 0.15, and 0.10. This generates graphs with a power-law degree distribution, and the out-degree of the most connected vertex is $\mathrm{O}(n^{0.6})$. We also assign uniform random time-stamps to edges for our experimental study.

We present performance results primarily on two single-socket Sun multithreaded servers, a Sun Fire T5120 system with the UltraSPARC T2 processor, and a Sun Fire 2000 system with the UltraSPARC T1 processor. The Sun UltraSPARC T2 is an eight-core processor, and the second-generation chip in Sun's Niagara architecture family. Each core is dual-issue and eight-way hardware multithreaded. Further, the hardware threads within a core are grouped into two sets of four threads each. There are two integer pipelines within a core, and each set of four threads share an integer pipeline. Each core also includes a floating point unit and a memory unit that are shared by all eight threads. Although the cores are dual-issue, each thread may only issue one instruction per cycle and so resource conflicts are possible. In comparison to the T2, the UltraSPARC T1 (the T2's predecessor) has only 4 threads per core, one integer pipeline per core, and one floating point unit shared by all eight cores. The 1.2 GHz UltraSPARC T2 cores share a 4 MB L2 cache, and the server has a main memory of 32 GB. The UltraSPARC T1 has 8 cores running at 1 GHz. The cores share a 3 MB L2 cache, and the main memory size is 16 GB. We build our codes with the Sun Studio 12 C compiler on the UltraSPARC T2 with the following optimization flags: *-fast -xtarget=ultraT2 -xchip=ultraT2 -xipo -m64 -xopenmp -xpagesize=4M*, and with Studio 11 on the UltraSPARC T1 with flags: *-fast -xtarget=ultraT1 -xchip=ultraT1 -xipo -xarch=v9b -xopenmp -xpagesize=4M*.

Our implementations can be easily ported to other multicore and symmetric multiprocessor (SMP) platforms. We also execute experiments on an IBM pSeries SMP system, the Power 570. The Power 570 is a 16-way symmetric multiprocessor of 1.9 GHz Power5 processors with simultaneous multithreading (SMT), a 32 MB shared L3 cache, and 256 GB shared memory.

2. Graph Representation

We represent the interaction data as a graph $G = (V, E)$, where V is the set of vertices representing unique interacting entities, and E is the set of edges representing the interactions. The number of vertices and edges are denoted by n

and m, respectively. The graph can be directed or undirected, depending on the input interaction data set. We will assume that each edge $e \in E$ has a positive integer weight $w(e)$. For unweighted graphs, we use $w(e) = 1$. A *path* from vertex s to t is defined as a sequence of edges $\langle u_i, u_{i+1} \rangle$, $0 \leq i < l$, where $u_0 = s$ and $u_l = t$. The *length* of a path is the sum of the weights of edges. We use $d(s, t)$ to denote the distance between vertices s and t (the minimum length of any path connecting s and t in G). Let us denote the total number of shortest paths between vertices s and t by σ_{st}, and the number of those passing through vertex v by $\sigma_{st}(v)$.

We model dynamic interaction data by augmenting a static graph G with explicit time-ordering on its edges. In addition, vertices can also store attributes with temporal information, if required. Formally, we model dynamic networks as defined by Kempe et al. [17]: a temporal network is a graph $G(V, E)$ where each edge $e \in E$ has a *time-stamp* or *time label* $\lambda(e)$, a non-negative integer value associated with it. This time-stamp may vary depending on the application domain and context: for instance, it can represent the time when the edge was added to the network in some cases, or the time when two entities last interacted in others. If necessary, we can define multiple time labels per edge. We can similarly define time labels $\xi(v)$ for vertices $v \in V$, capturing, for instance, the time when the entity was added or removed. It is straight-forward to extend a static graph representation to implement time-stamps defined in the above manner. Also, the time-stamps can be abstract entities in the implementation, so that they can be used according to the application requirement.

2.1. Adjacency Structures

Efficient data structures and representations are key to high performance dynamic graph algorithms. In order to process massive graphs, it is particularly important that the data structures are space-efficient (compact). Ease of parallelization and the synchronization overhead also influence our representation choice. Ideally, we would like to use simple, scalable and low-overhead (for instance, lock-free) data structures. Finally, the underlying algorithms and applications dictate our data structure choices.

For representing sparse static graphs, adjacency lists implemented using cache-friendly adjacency arrays are shown to give substantial performance improvements over linked-list and adjacency matrix representations for sparse graphs [22]. For dynamic networks, we need to process insertions of vertices and edges, which may be streaming or batched. Thus, we experiment with the performance of several candidate representations for small-world graphs and discuss the trade-offs associated with each data structure.

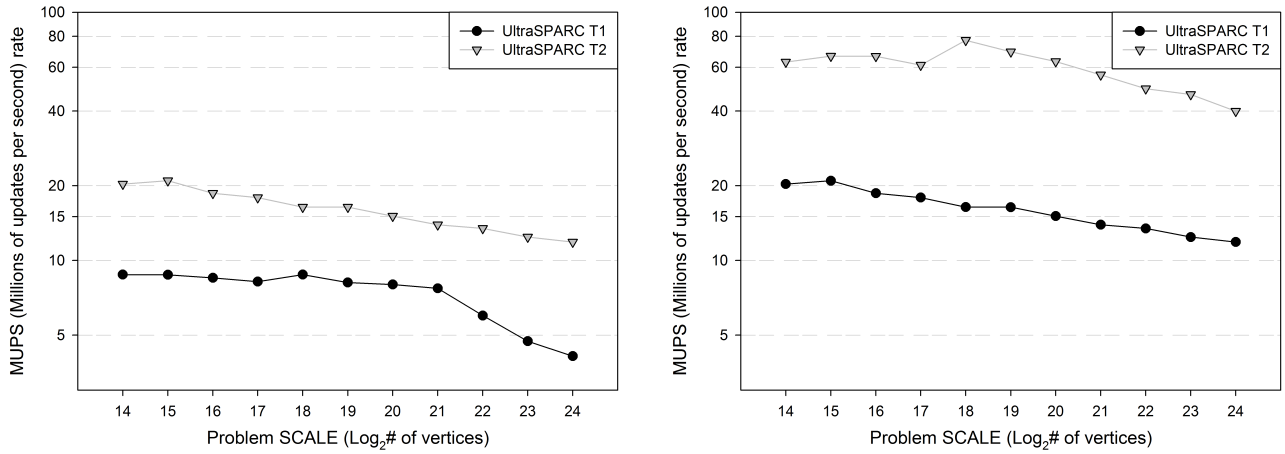
2.1.1. Dynamic Arrays (*Dyn-arr*). We can extend the static graph representation and use dynamic, resizable adjacency arrays for processing insertions and deletions. Clearly, this would support fast insertions. For inserting a new edge $\langle u, v \rangle$, we resize the adjacency array of u if needed, and place v at the end of the array. The count can be incremented using an atomic increment operation on modern hardware, and so this representation supports lock-free, non-blocking insertions.

There are two potential parallel performance issues with this data structure. Edge deletions are expensive in this representation, as we may have to scan the entire adjacency list in the worst case to locate the required tuple. Note that a scan is reasonable for low-degree vertices (say, less than 10 adjacencies), given the spatial locality we have due to the contiguous adjacency representation. However, due to the power-law degree distributions in small-world networks, we may encounter several high-degree vertices in the graph, and deletions for these vertices would necessitate $\mathrm{O}(n)$ additional work.

The second issue is with load balancing of parallel edge insertions among the processors. We might be faced with a bad scenario of several threads trying to increment the count of a high-degree vertex simultaneously, if there are a stream of contiguous insertions corresponding to adjacencies of one vertex. This can be addressed by batching the updates, or by randomly shuffling the updates before scheduling the insertions. Both these operations incur an additional overhead, but since insertions are still constant-time operations, the performance hit is not as significant as other data structures.

There is also the performance overhead associated with resizing the adjacency arrays frequently. We implement our own memory management scheme by allocating a large chunk of memory at the algorithm initiation, and then have individual processors access this memory block in a thread-safe manner as they require it. This avoids frequent system malloc calls. Since we assume that SNAP is analyzing power-law graphs, we double the size of the adjacency array every time we resize it. There is still a trade-off associated with the memory footprint and the number of array resizes. We experiment with various initial sizes and resizing heuristics, and pick a scheme that is easy to implement and deterministic. We set the size of each adjacency array to km/n initially, and we find that a value of $k = 2$ performs reasonably well on our test instances. When analyzing performance, we also report execution time for a *optimal*-case array representation *Dyn-arr-nr*, which assumes that one knows the size of the adjacency arrays for each vertex before-hand, and thus incurs no resizing overhead.

The problem size also influences the graph representation that one might choose. Figure 1 gives the parallel performance achieved with *Dyn-arr-nr* for a series of insertions on one core (left), and eight cores (right) of the UltraSPARC T1 and UltraSPARC T2 systems. We report update performance in terms of a performance rate MUPS (millions of updates



(a) 1 core: 4 threads on UltraSPARC T1 and 8 threads on UltraSPARC T2.

(b) 8 cores: 32 threads on UltraSPARC T1 and 64 threads on UltraSPARC T2.

Figure 1. Parallel performance of insertions with the *Dyn-arr-nr* representation as the size of the problem instance (synthetic R-MAT graphs, $m = 10n$) is varied from thousands to tens of millions of vertices.

per second), which is the number of insertions or deletions divided by the execution time in seconds. We generate synthetic R-MAT networks with $m = 10n$ edges, and vary n across three orders of magnitude. Observe that UltraSPARC T2 performance on 8 cores drops by a factor of 1.5, and UltraSPARC T1 performance by a factor of 1.8, as we vary n from 2^{14} to 2^{24}. When the memory footprint of the run is comparable to the $L2$ cache size, the performance is relatively higher. For the rest of the paper, we will be focusing on graph instances with a memory footprint significantly larger than the $L2$ cache size.

In Figure 2, we plot the performance overhead of array resizing for a large-scale graph. We generate the dynamic network through a series of insertions, and report the MUPS rate as the number of threads is varied from 1 to 64. The initial array size is set to 16 in this case. Observe that the impact of resizing is not very pronounced in this case, which is a positive sign for the use of resizable arrays.

2.1.2. Batched Insertions and Deletions. In cases where the number of tuples to be inserted or deleted is comparatively high, an intuitive strategy is to order the tuples by the vertex identifier and then process all the updates corresponding to each vertex at once. This is a better solution to the load-balancing problem we discussed in the previous representations. The time taken to semi-sort updates by their vertex is a lower bound for this strategy. Sorting can be done in parallel, and would require $O(k)$ work (for a batch of k updates).

2.1.3. Vertex (*Vpart*) and Edge (*Epart*) partitioning. The overhead due to locking is a performance bottleneck in the parallel implementation using *Dyn-arr*. To avoid locking, one strategy would be to assign vertices to processors in a randomized or a deterministic fashion (we call this representation *Vpart*) so that threads can insert to different vertices simultaneously. However, each update is read by all the threads in this case, which can be significant additional work. The reads have good spatial locality, and hence this approach might work well for a small number of threads.

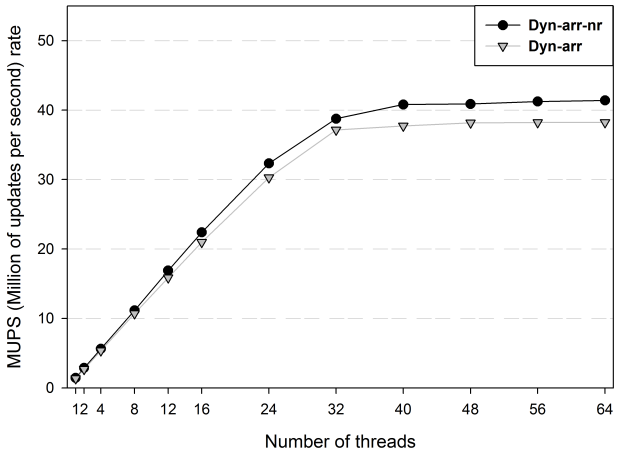


Figure 2. A comparison of UltraSPARC T2 parallel performance of *Dyn-arr* and *Dyn-arr-nr* for graph construction, represented as a series of insertions. We generate an R-MAT network of 33.5 million vertices and 268 million edges.

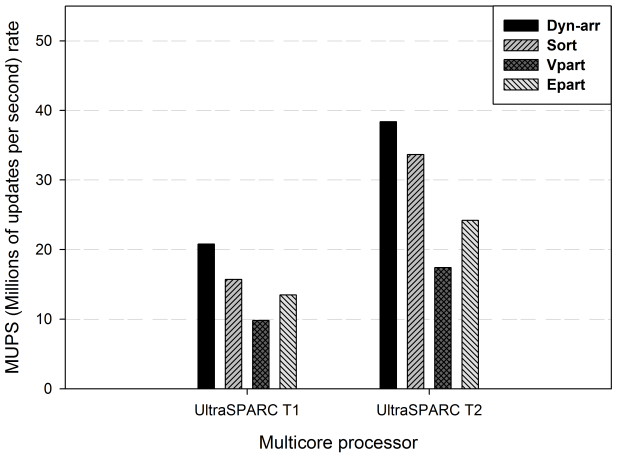


Figure 3. An upper bound on the performance achieved using batched operations for insertions (semi-sorting time), compared to the performance of *Dyn-arr-nr*. These are parallel performance results on UltraSPARC T2.

Similarly, we can also partition the adjacency lists of vertices discovered to be *high-degree* ones in the process of insertions. This would help us avoid the load-balancing problem when there are multiple insertions to the adjacency of a single vertex. We call this implementation *Epart*. A drawback with the *Epart* implementation is the space overhead of splitting up the adjacency list of high-degree vertices among multiple threads, and the subsequent merge step to construct a single adjacency list representation.

Figure 3 plots the performance of insert-only updates with *Dyn-arr*, and compares it with the performance that can be achieved using vertex partitioning, edge partitioning, and batched processing (the upper bound of semi-sorting) on 8 cores of UltraSPARC T2 and UltraSPARC T1, for a graph of 33.5 million vertices and 268 million edges. We consider the semi-sorting time to obtain an upper bound for the MUPS score that can be achieved with a batched representation. *Dyn-arr* outperforms the batched representation, as well as *Epart* and *Vpart*. The trends on UltraSPARC T2 and UltraSPARC T1 are similar.

2.1.4. Treaps. The resizable array data structure supports fast access and insertions, but deletions can be very expensive. For applications in which the frequency of edge deletions is high, we consider alternate representations. A possible choice for representing the adjacency list would be a self-balancing binary tree structure such as an AVL-tree or a Red-Black tree. These data structures allow for fast searches, insertions, and deletions with worst-case $O(\log n)$ update time. We chose to use a simpler self-balancing data structure, the treap [23], to represent the adjacencies.

Treaps are binary search trees with a priority (typically a random number) associated with each node. The priorities are maintained in heap order, and this data structure supports insertions, deletions, and searching in average-case $O(\log n)$ time. In addition, there are efficient parallel algorithms for set operations on treaps such as union, intersection and difference. Set operations are particularly useful to implement kernels such as graph traversal and induced sub-graphs, and for batch-processing updates.

Since we use our own memory management routines, our *Treaps* implementation is reasonably space-efficient, but with a 2-4 times larger memory footprint compared to *Dyn-arr*. We use the same resizing strategies for *Treaps* as with *Dyn-arr*. Insertions are significantly slower than *Dyn-arr*. Note that we cannot atomically increment the size counter for updates, as the treap may undergo rebalancing at every step. The granularity of work inside a lock is significantly higher in this case, and so the problem of multiple threads simultaneously trying to insert to a vertex is also significant. We can overcome this by processing the insertions in batches, but randomly shuffling the tuples might not be as effective as in the case of *Dyn-arr*.

In case of deletions, however, this representation has a significant performance advantage. We actually remove the node out of the adjacency array in case of *Treaps*, whereas we just mark a memory location as deleted for *Dyn-arr*.

2.1.5. A hybrid representation (*Hybrid-arr-treap*). To process both insertions and deletions efficiently, and also given the power-law degree distribution for small-world networks, we design a hybrid representation that uses *Dyn-arr* to represent adjacencies of *low-degree* vertices, and *Treaps* for *high-degree* vertices. We monitor a parameter *degree-thresh* that decides which representation to use for the vertex. By using *Dyn-arr* for low-degree vertices (which will be a majority of vertices in the graph), we can achieve good performance for insertions. Also, deletions are fast for low-degree vertices, whereas they take logarithmic time for high-degree vertices represented using treaps. Thus we are able to optimize for good performance for both insertions as well as deletions using this data structure. Based on the value of *degree-thresh*, we can change the representation of an adjacency list from one to the other. We have experimented with several different values and find that for synthetic R-MAT small-world graphs, a value of 32 on our platforms provides a reasonable insertion-deletion performance trade-off for an equal number of insertions and deletions. Given the graph update rate and the insertion to deletion ratio for an application, it may be possible to develop runtime heuristics for a reasonable threshold.

Figures 4 and 5 plot the parallel performance of insertions and deletions respectively, on a large-scale synthetic R-MAT instance of 33.5 million vertices and 268 million edges on UltraSPARC T2. The insertion MUPS rate is computed from

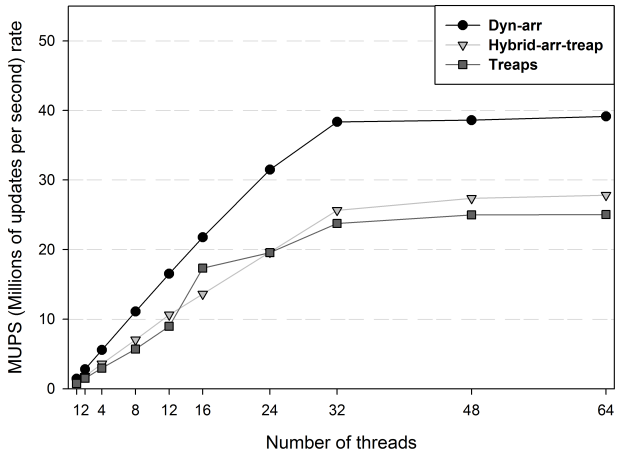


Figure 4. A comparison of UltraSPARC T2 parallel performance using the *Dyn-arr*, *Treaps*, and *Hybrid-arr-treap* representations, for graph construction (treated as a series of insertions). We generate an R-MAT network of 33.5 million vertices and 268 million edges.

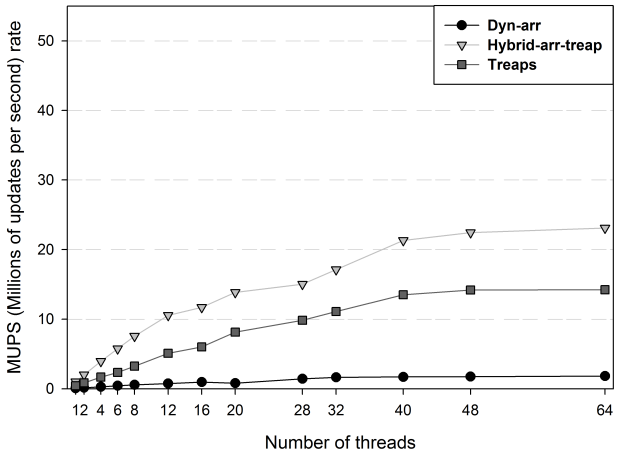


Figure 5. A comparison of UltraSPARC T2 parallel performance for deletes using the *Dyn-arr*, *Treaps*, and *Hybrid-arr-treap* representations. We generate an R-MAT network of 33.5 million vertices and 268 million edges, and compute execution time for 20 million deletions.

the time taken to construct the graph, while the deletion rate is the normalized value for 20 million random deletions after constructing this network. *Dyn-arr* is 1.4 times faster than the hybrid representation, while *Hybrid-arr-treap* is slightly faster than *Treaps*. The real benefit of using the hybrid representation is seen for deletions, where *Hybrid-arr-treap* is almost 20× faster than the dynamic array representation. *Hybrid-arr-treap* is also significantly faster than *Treaps* in case of deletions. In Figure 6, we plot the parallel performance of updates given a mix of insertions and deletions. We construct a synthetic network of 33.5 million vertices and 268 million edges, and compute the average execution time for a random selection of 50 million updates, with 75% insertions and 25% deletions. We find that the performance of *Hybrid-arr-treap* and *Dyn-arr* are comparable in this case, while *Treaps* is slower. For a large proportion of deletions, the performance of *Hybrid-arr-treap* would be better than *Dyn-arr*.

2.1.6. Other optimizations. Compressed graph structures are an attractive design choice for processing massive networks, and they have been extensively studied in the context of web-graphs [7]. Exploiting the small-world and self-similarity properties of the web-graph, mechanisms such as vertex reordering, compact interval representations, and compression of similar adjacency lists have been proposed. It is an open question on how these techniques perform for real-world networks from other applications, and whether they can be extended for processing dynamic graphs.

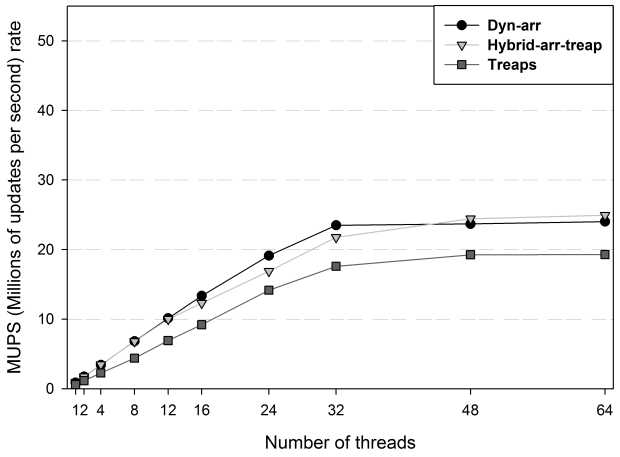


Figure 6. A comparison of UltraSPARC T2 parallel performance for a combination of insertions and deletes using the *Dyn-arr*, *Treaps*, and *Hybrid-arr-treap* representations. We generate an R-MAT network of 33.5 million vertices and 268 million edges, and compute execution time for 50 million updates.

3. Graph Analysis Kernels

We next identify key connectivity, path-related, and centrality-related graph *kernels* (or algorithmic building blocks) in the design of higher-level analysis approaches) for analyzing dynamic networks, present fast parallel algorithms

for solving them, and evaluate their performance on parallel architectures.

3.1. Connectivity

Consider the following problem of graph connectivity: *given two vertices s and t in a graph, determine whether there is a path connecting them*. A possible approach to solving this problem would be to do a breadth-first graph traversal from s and determine if we can reach t. However, we can process queries on path existence (where we do not need the length of the path) more efficiently by maintaining a spanning forest corresponding to the graph. In case of dynamic networks, maintaining a forest that changes over time with edge insertions and deletions is known as the *dynamic forest* problem, and this is a key data structure in several dynamic graph and network flow problems [27]. There are well-known data structures for dynamic trees, which rely on path decomposition (e.g., link-cut trees) or tree contraction (e.g., RC-trees, top trees).

The link-cut tree is a data structure representing a rooted tree [24]. The basic structural operations are *link*(v,w), which creates an arc from a root v to a vertex w, and *cut*(v), which deletes the arc from v to its parent. The routines *findroot*(v) and *parent*(v) can be used to query the structure of the tree. There are several possible implementations of the link-cut tree, including self-adjusting ones which guarantee a logarithmic query time. A straightforward implementation of the link-cut tree would be to store with each vertex a pointer to its parent. This supports the *link*, *cut*, and *parent* in constant time, but the *findroot* operation would require a worst-case traversal of O(n) vertices for an arbitrary tree. However, we observe that for low-diameter graphs such as small-world networks, this operation just requires a small number of hops, as the height of the tree is small. Also, it is relatively simple to construct the link-cut tree, given the network. We apply a lock-free, level-synchronous parallel breadth-first search [4] on the graph to produce a tree associated with the largest component in the network, and then run connected components to construct a forest of link-cut trees. Figure 7 plots the execution time and parallel speedup achieved on the UltraSPARC T2 system for constructing the link-cut tree corresponding to a synthetic small-world network of 10 million vertices and 84 million edges. We achieve a good speedup of 22 on 32 threads of the UltraSPARC T2 system. Figure 8 plots the execution time for 1 million connectivity queries using this link-cut tree. Each connectivity query involves two *findroot* operations, each of which would take O(d) time (where d is the diameter of the network). The queries can be processed in parallel, as they only involve memory reads. Each query computation is essentially a linked list traversal, the serial performance of which is poor on cache-based architectures. We still achieve a speedup of 20 for parallel query processing on UltraSPARC T2.

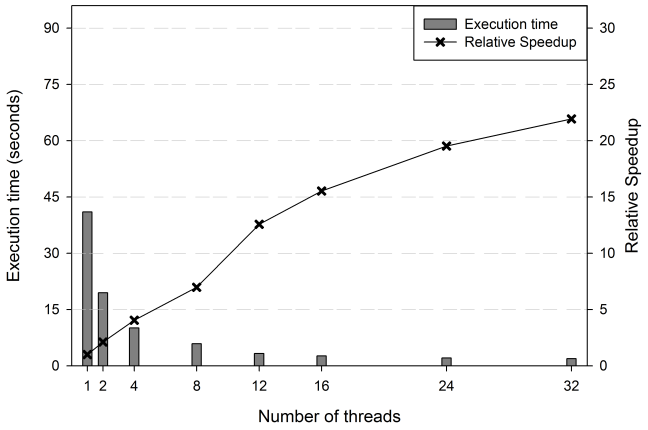


Figure 7. Execution time and parallel speedup for link-cut tree construction (from a small-world network of 84 million edges) on UltraSPARC T2.

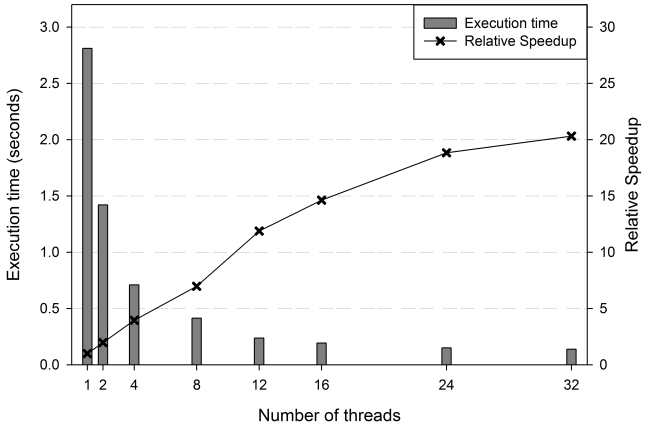


Figure 8. Execution time and parallel speedup on UltraSPARC T2 for 1 million connectivity queries (link-cut tree, small-world network of 84 million edges).

3.2. Induced subgraph

Utilizing temporal information, several dynamic graph problems can be reformulated as problems on static instances. For instance, given edge and vertex time labels, we may need to extract vertices and edges created in a particular time interval, or analyze a snapshot of a network. This problem can be solved by the induced subgraph kernel, and the execution time is dependent on the size and connectivity of the vertex or edge set that induce the subgraph.

Figure 9 plots the parallel performance of the induced subgraph kernel on UltraSPARC T1. We apply the induced

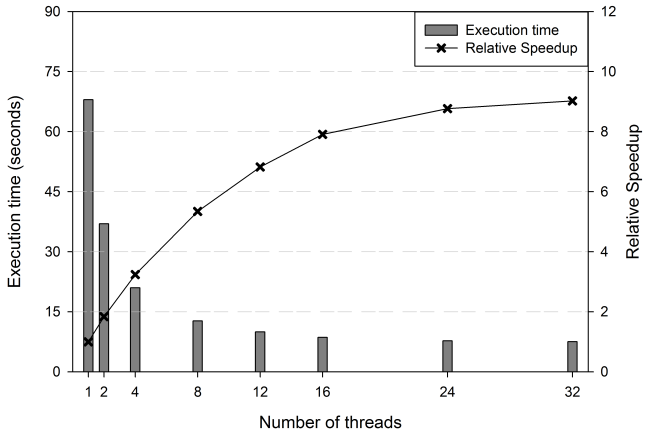


Figure 9. Parallel Performance of the induced subgraph kernel on UltraSPARC T1 for an R-MAT graph of 20 million vertices and 200 million edges.

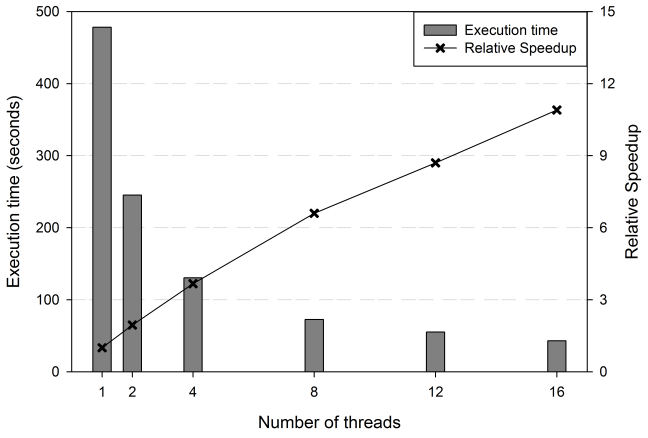


Figure 10. Parallel BFS performance on the IBM Power 570 system for an R-MAT graph of 500 million vertices and 4 billion edges.

subgraph kernel on a graph of 20 million vertices and 200 million edges, with each edge assigned a integral time-stamp between 1 and 100 during graph creation. The edges are randomly shuffled to remove any locality associated due to graph generation. We generate the induced subgraph corresponding to the edges inserted in time interval (20, 70). The first step in the algorithm is to identify edges that are affected by the temporal condition we apply. This step just requires us to make one pass over the edge list and mark all the affected edges, and also keep a running count. Next we either create a new graph, or delete edges from the current graph, depending on the the affected edge count. This step reduces to the case of insertions and deletions of edges to the graph, which we discussed in the previous section. Thus, each edge in the graph is visited at most twice in the worst case in this kernel. As demonstrated in Figure 9, the induced subgraph kernel achieves a good parallel speedup on UltraSPARC T1.

3.3. Graph traversal

Graph traversal is a fundamental technique used in several network algorithms. Breadth-first search (BFS) is an important approach for the systematic exploration of large-scale networks. In prior work, we designed a level-synchronous PRAM algorithm for BFS that takes $O(d)$ time and optimal linear work (where d is the graph diameter) [4]. For small-world graphs, where d is typically a constant, or in some cases $O(\log n)$, this algorithm can be efficiently implemented on large shared memory parallel systems with a high parallel speedup.

For isolated runs of BFS on dynamic graphs, we can take the approach discussed in the induced subgraph kernel, i.e., utilize the time-stamp information and recompute from scratch. This approach requires no additional memory, as it just uses the time label information for filtering vertices and edges during graph traversal. Coupled with optimizations to handle graphs with unbalanced degree distributions, we are able to traverse massive graphs in just a few seconds on current HPC systems. For instance, we generate a massive synthetic small-world network with 500 million vertices and 4 billion edges, with time-stamps on edges such that the entire graph is in one giant component. On 16 processors of the IBM Power 570 SMP, augmented BFS with a check for time-stamps takes just 46 seconds for this large-scale network (see Figure 10 for parallel performance and scaling). The unbalanced degree optimization we refer to above are discussed in more detail in [4, 5]. We process the high-degree and low-degree vertices differently in a parallel phase to ensure that each that we have balanced partitioning of work to threads.

3.4. Temporal path-based centrality metrics

Finally, we present the case study of a social network analysis routine that can be parallelized efficiently using the data structures and kernels presented so far – *centrality analysis* in interaction networks.

A fundamental problem in social network analysis is to determine the *importance* (or the *centrality*) of a particular vertex (or an edge) in a network. Well-known quantitative metrics for computing centrality are closeness, stress, and betweenness. Of these indices, betweenness has been extensively used in recent years for the analysis of social-interaction networks, as well as other large-scale complex networks. Some applications include assessing lethality in

biological networks [16], study of sexual networks and AIDS, identifying key actors in terrorist networks [9], organizational behavior, and supply chain management processes.

Betweenness centrality can be formulated for entities in a dynamic network also, by taking the interaction time labels and their ordering into consideration. Define a temporal path $p_d\langle u, v\rangle$ [17] between u and v as a sequence of edges $\langle u = v_0, v_1\rangle$, $\langle v_1, v_2\rangle$, ..., $\langle v_{k-1}, v_k = v\rangle$, such that $\lambda(v_{t-1}, v_t) < \lambda(v_t, v_{t+1})$ for $t = 1, 2, ..., k-1$. We define a *temporal shortest path* between u and v as a temporal path with the shortest distance $d(u, v)$. In this framework, the temporal betweenness centrality $BC_d(v)$ of a vertex v is the sum of the fraction of all temporal shortest paths passing through v, between all pairs of vertices in the graph. Formally, let $\delta_{st}(v)$ denote the pairwise dependency, or the fraction of shortest temporal paths between s and t that pass through v: $\delta_{st}(v) = \frac{\sigma_{st}(v)}{\sigma_{st}}$. Then, $BC_d(v) = \sum_{s \neq v \neq t \in V} \delta_{st}(v)$. Note that a shortest temporal path between u and v may not necessarily be a shortest path in the aggregated graph formed by ignoring time-stamps. This definition of temporal centrality respects the time ordering of edges and provides better insight into network evolution. Alternate definitions of temporal paths are also possible. For instance, one could define a *valid edge* in the graph to fall within a predefined range of timestamps.

We design and implement a new parallel approach for computing temporal betweenness centrality, by augmenting our prior parallel algorithm for betweenness computation on static graphs [5] with time-stamp information. The graph traversal step in this parallel approach is modified to process temporal paths, while the dependency-accumulation stage remains unchanged. Figure 11 plots the performance of approximate betweenness centrality (graph traversal from only a subset of vertices, and then extrapolation of the centrality scores) for a synthetic network with 33 million vertices and 268 million edges. We achieve a speedup of 23 on 32 threads of UltraSPARC T2 for this particular problem instance. We assign integer time-stamps in the interval $[0, 20]$ for the vertices, and follow the notion of temporal shortest paths defined above. We traverse the graph from 256 randomly chosen vertices and then approximate the betweenness values. It is straightforward to modify our implementation to process other edge filtering or temporal path conditions as well. In addition to picking the shortest path, edges are filtered in every phase of the graph traversal. Thus, the amount of concurrency per phase is comparatively lower than breadth-first graph traversal with time-stamps.

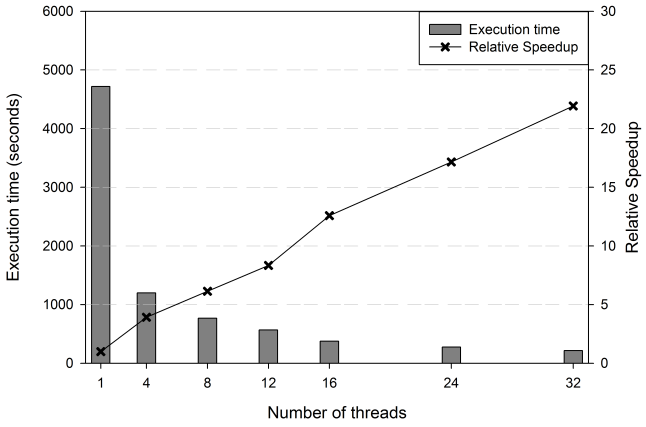


Figure 11. Approximate betweenness performance on UltraSPARC T2 using the notion of temporal paths. The input network is a synthetic R-MAT graph of 33 million vertices and 268 million edges.

4. Conclusions and Future Research

We present the first work on the design and implementation of high-performance graph representations and kernels for analyzing massive dynamic interaction data sets with billions of entities. In particular, we present a hybrid graph representation for small-world graphs with unbalanced degree distributions, that processes both insertions and deletions of edges efficiently. We design a fast parallel implementation of the link-cut tree for connectivity queries. Finally, we report impressive parallel performance results for several graph traversal and path-based dynamic kernels on current multicore and SMP systems.

There are several challenging problems in this area for future research. We intend to explore compressed adjacency representations to reduce the memory footprint, and vertex and edge identifier reordering strategies to improve cache performance. The problem of single-source shortest paths for arbitrarily weighted graphs is challenging to parallelize efficiently, and is even harder in a dynamic setting. We will study how other static complex graph algorithms can be modified to work with our data representation. We will also investigate whether the data structures and parallelization techniques discussed in this paper can be applied to graph analysis on heterogeneous multicore processors such as the IBM Cell Broadband Engine processor, and massively multithreaded systems such as the Cray XMT.

Acknowledgments

This work was supported in part by NSF Grants CNS-0614915 and DBI-0420513, IBM Faculty Fellowship and Microsoft Research grants, NASA grant NP-2005-07-375-HQ, DARPA Contract NBCH30390004, PNNL CASS-MT Center, MIT Lincoln Laboratory, and by the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. We acknowledge the support from Sun Microsystems with

their donation of Sun Niagara blades through an Academic Excellence Grant. We thank Bruce Hendrickson and Jon Berry of Sandia National Laboratories, and Jeremy Kepner of MIT Lincoln Laboratory for discussions on large-scale graph problems.

References

[1] L. Amaral, A. Scala, M. Barthélémy, and H. Stanley, "Classes of small-world networks," *Proc. National Academy of Sciences USA*, vol. 97, no. 21, pp. 11 149–11 152, 2000.

[2] D. Bader and G. Cong, "Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs," in *Proc. Int'l Parallel and Distributed Processing Symp. (IPDPS 2004)*, Santa Fe, NM, Apr. 2004.

[3] D. Bader, G. Cong, and J. Feo, "On the architectural requirements for efficient execution of graph algorithms," in *Proc. 34th Int'l Conf. on Parallel Processing (ICPP)*. Oslo, Norway: IEEE Computer Society, Jun. 2005.

[4] D. Bader and K. Madduri, "Designing multithreaded algorithms for breadth-first search and st-connectivity on the Cray MTA-2," in *Proc. 35th Int'l Conf. on Parallel Processing (ICPP)*. Columbus, OH: IEEE Computer Society, Aug. 2006.

[5] D. Bader and K. Madduri, "Parallel algorithms for evaluating centrality indices in real-world networks," in *Proc. 35th Int'l Conf. on Parallel Processing (ICPP)*. Columbus, OH: IEEE Computer Society, Aug. 2006.

[6] D. Bader and K. Madduri, "SNAP, Small-world Network Analysis and Partitioning: an open-source parallel graph framework for the exploration of large-scale networks," in *Proc. Int'l Parallel and Distributed Processing Symp. (IPDPS 2008)*, Miami, FL, Apr. 2008.

[7] P. Boldi and S. Vigna, "The WebGraph framework I: compression techniques," in *Proc. 13th Intl. Conf. on World Wide Web (WWW 13)*. New York, NY, USA: ACM Press, 2004, pp. 595–602.

[8] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining," in *Proc. 4th SIAM Intl. Conf. on Data Mining (SDM)*. Orlando, FL: SIAM, Apr. 2004.

[9] T. Coffman, S. Greenblatt, and S. Marcus, "Graph-based technologies for intelligence analysis," *Communications of the ACM*, vol. 47, no. 3, pp. 45–47, 2004.

[10] C. Demetrescu, I. Finocchi, and G. Italiano, "Dynamic graphs," in *Handbook on Data Structures and Applications*, D. Mehta and S. Sahni, Eds. CRC Press, 2005, ch. 36.

[11] C. Demetrescu, I. Finocchi, and A. Ribichini, "Trading off space for passes in graph streaming problems," in *Proc. 17th Ann. Symp. Discrete Algorithms (SODA-06)*. Miami, FL: ACM Press, Jan. 2006, pp. 714–723.

[12] D. Eppstein, Z. Galil, and G. Italiano, "Dynamic graph algorithms," in *Handbook of Algorithms and Theory of Computation*, M. Atallah, Ed. CRC Press, November 1998, ch. 8.

[13] D. Eppstein, Z. Galil, G. Italiano, and A. Nissenzweig, "Sparsification: a technique for speeding up dynamic graph algorithms," *J. ACM*, vol. 44, no. 5, pp. 669–696, 1997.

[14] M. Henzinger and V. King, "Randomized dynamic graph algorithms with polylogarithmic time per operation," in *Proc. 27th Ann. Symp. of Theory of Computing (STOC)*. New York, NY, USA: ACM Press, 1995, pp. 519–527.

[15] M. Henzinger, P. Raghavan, and S. Rajagopalan, "Computing on data streams," Compaq Systems Research Center, Palo Alto, CA, Tech. Rep. TR-1998-011, May 1998.

[16] H. Jeong, S. Mason, A.-L. Barabási, and Z. Oltvai, "Lethality and centrality in protein networks," *Nature*, vol. 411, pp. 41–42, 2001.

[17] D. Kempe, J. Kleinberg, and A. Kumar, "Connectivity and inference problems for temporal networks," *J. Comput. Syst. Sci.*, vol. 64, no. 4, pp. 820–842, 2002.

[18] V. Krebs, "Mapping networks of terrorist cells," *Connections*, vol. 24, no. 3, pp. 43–52, 2002.

[19] K. Madduri, D. Bader, J. Berry, and J. Crobak, "An experimental study of a parallel shortest path algorithm for solving large-scale graph instances," in *Proc. The 9th Workshop on Algorithm Engineering and Experiments (ALENEX07)*, New Orleans, LA, Jan. 2007.

[20] S. Muthukrishnan, "Data streams: algorithms and applications," *Foundations and Trends in Theoretical Computer Science*, vol. 1, no. 2, pp. 117–236, 2005.

[21] M. Newman, "The structure and function of complex networks," *SIAM Review*, vol. 45, no. 2, pp. 167–256, 2003.

[22] J. Park, M. Penner, and V. Prasanna, "Optimizing graph algorithms for improved cache performance," in *Proc. Int'l Parallel and Distributed Processing Symp. (IPDPS 2002)*. Fort Lauderdale, FL: IEEE Computer Society, Apr. 2002.

[23] R. Seidel and C. Aragon, "Randomized search trees," *Algorithmica*, vol. 16, pp. 464–497, 1996.

[24] D. Sleator and R. Tarjan, "A data structure for dynamic trees," *J. Comput. Syst. Sci.*, vol. 26, no. 3, pp. 362–391, 1983.

[25] Y. Tian, R. McEachin, C. Santos, D. States, and J. Patel, "SAGA: A subgraph matching tool for biological graphs," *Bioinformatics*, vol. 23, no. 2, pp. 232–239, 2007.

[26] D. Watts and S. Strogatz, "Collective dynamics of small world networks," *Nature*, vol. 393, pp. 440–442, 1998.

[27] R. Werneck, "Design and analysis of data structures for dynamic trees," Ph.D. dissertation, Princeton University, Princeton, June 2006.

[28] C. Zaroliagis, "Implementations and experimental studies of dynamic graph algorithms," in *Experimental algorithmics: from algorithm design to robust and efficient software*. New York, NY, USA: Springer-Verlag New York, Inc., 2002, pp. 229–278.